ADAPTING LINUX VFAT FILESYSTEM TO EMBEDDED OPERATING SYSTEMS

Hung-Kai Ting[ξ], *Ching-Ru Lo*[+], *Mei-Ling Chiang*[+], *and Ruei-Chuan Chang*[ξ]

Department of Information Management[+]
National Chi-Nan University, Puli, Taiwan, R.O.C.
Email: s0213523@ncnu.edu.tw, joanna@ncnu.edu.tw

Department of Computer and Information Science[ξ]
National Chiao-Tung University, Hsinchu, Taiwan, R.O.C.
Email: rc@cc.nctu.edu.tw

ABSTRACT

Nowadays, embedded systems play an important role in the new living fashion. Many embedded systems need storage capability to offer advanced application features. However, to implement file system for a target operating system from the scratch is a time-consuming and error-prone task. Linux under the spirit of GNU GPL and open source codes gains its stability, reliability, and high performance. Therefore, making use of the existing Linux vfat FileSystem source codes to adapt to embedded systems becomes a feasible and cost-effective way.

This paper describes how to adapt Linux vfat FileSystem to LyraOS, a component-based operating system for embedded systems. Under different system design principles and kernel architectures, this work includes remodeling and modifying Linux vfat FileSystem to be a separate and self-contained C++ component, replacing Linux kernel support functions invocation with equivalent LyraOS kernel components invocation, adding data path to RAM-based device, and implementing compatible file system interfaces (POSIX) for LyraOS vfat filesystem.

Performance evaluation under modified Andrew BenchMark shows that our LyraOS vfat FileSystem operates at low cost. The experience of this study can be of practical value to serve as the reference for embedding Linux file system into a target system that needs storage capability.

KEYWORDS

File System, Linux vfat FileSystem, Embedded System, Component-based

1. INTRODUCTION

Embedded applications are versatile and the hardware devices range from simple controllers to more complex systems. For the versatile hardware devices and different application requirements, a reconfigurable embedded operating system is needed. Thus, various operating systems design dedicated for embedded systems are thus created, such as PalmOS [22], EPOC [12], Windows CE [25], GEOS [16], QNX [23], Pebble [2,17], MicroC/OS [21], eCos [11], LyraOS [3-7,18,19,26], etc.

Many embedded systems, such as Digital Cameras, PDAs, and MP3 players need storage capability to take advantage of the many advanced application features. File System is the key component for storing data in embedded systems, which organizes, manages and maintains the file hierarchy into mass-storage device.

Because of the spirit of GNU General Public License (GPL) [15] and open source codes, Linux [1] gains its popularity and has the advantages of stability, reliability, high performance, and well documentation. These advantages let making use of the existing open source codes and adapting Linux File System for target operating system becomes a feasible and cost-effective way. Although Linux supports lots of file systems, Linux vfat FileSystem which is compatible with the popular Microsoft FAT File System and supports long file name is preferable, in this paper, to adapt to embedded systems.

With the falling cost of SDRAM and its growing storage capacity, a transition from disk storages to RAM-based storage devices is happening in embedded systems [9]. Besides, due to data transfer delay of disk operations and the requirements of light weight, small size, low power consumption, or mobility, RAM-based storage

devices are deployed in most of embedded systems.

This paper describes how to adapt Linux vfat FileSystem to LyraOS [3-7,18,19,26]. LyraOS is a component-based operating system designed for embedded systems. Under the component design principle [2,14,17,20], the Linux vfat FileSystem should also be implemented as a separate component, such that the advantages of modularity, reconfigureability, component replacement and reuse can be maintained. However, there are many difficulties should be dealt with for this adaptation. For example, being a monolithic kernel, Linux vfat FileSystem is not a separate component that has closely relationship and interaction with the other kernel functions such as Virtual File System (VFS), buffer cache, device driver, and kernel core.

Therefore, this adaptation work should solve the difficulties from different system design principles and different kernel architectures. Our work focuses on three parts. First, implementing file system as a self-contained component which requires modifying the Linux vfat FileSystem codes to separate them from the other kernel functions. Second, replacing Linux kernel supported functions invocation with equivalent LyraOS kernel components invocation for providing kernel services for vfat FileSystem. Third, adding data paths to RAM-based storage devices.

The rest of this paper is organized as follows. Section 2 gives an overview the Linux vfat FileSystem, including how it cooperates with VFS (Virtual File System). Section 3 briefly introduces LyraOS. The difficulties for this adaptation are also discussed. Section 4 presents the adaptation work including remodeling and modifying Linux vfat FileSystem codes. Section 5 shows primitive performance evaluation results, and Section 6 concludes this paper.

2. LINUX VFAT FILESYSTEM

This section briefly introduces Linux File System architecture [27,28], Linux Virtual File System [29], and VFAT FileSystem.

2.1 Linux File System Architecture

Linux supports many types of file systems. In order to provide applications with a uniform programming interface (API), as shown in Table 1, Linux provides a common interface between user applications and those file systems. The Virtual File System (VFS) [27,28] is implemented in the kernel as the interface, which maintains those file systems to have a uniform programming interface for user applications. File systems follow VFS's exported common interface can be implemented as modules and be mounted for use when needed. The Linux File System Architecture is illustrated in Figure 1.

Table 1: Linux File System API

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
int link(const char *oldpath, const char *newpath);
int unlink(const char *pathname);
int chdir(const char *path);
int stat(const char *file_name, struct stat *buf);
off_t lseek(int fildes, off_t offset, int whence);
int rename(const char *oldpath, const char *newpath);
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
int ioctl(int d, int request, ...)
int truncate(const char *path, off_t length);
int chmod(const char *path, mode_t mode);
```

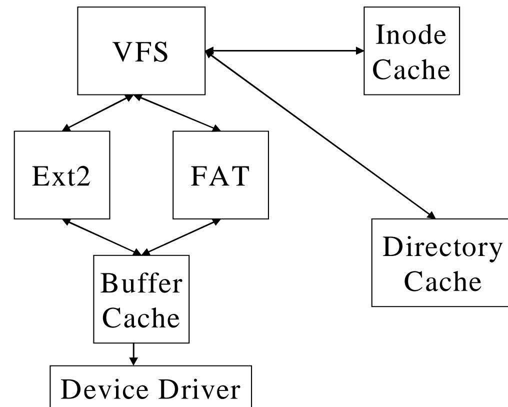


Figure 1: Linux File System Architecture

When file system system calls are invoked, *VFS* passes the requested services to the target mounted file system (i.e., Ext2, FAT,…). The target file system maps the logical data blocks to the physical data blocks, and then the *Device Driver* performs the I/O operations between file system and storage device. To speed up accesses, recently used device blocks are cached in the *Buffer Cache*. *Inode Cache* and *Directory Cache* are used to cache the recently opened file entries and directory entries information.

2.2 Linux vfat FileSystem Overview

The VFAT file system is the FAT file system with long file name support. The *FAT (File*

Allocation Table) is used much like a linked list. Table entry indexed by cluster number contains the cluster number of the next cluster in the file. Figure 2 shows an overview of the VFAT file system. When the *open("pathname")* system call is invoked, the *lookup()* function parses the pathname, from root to each entry name, and the *find()* function compares it with all entries under the parent entry to find if the file or directory entry is existent to be opened.

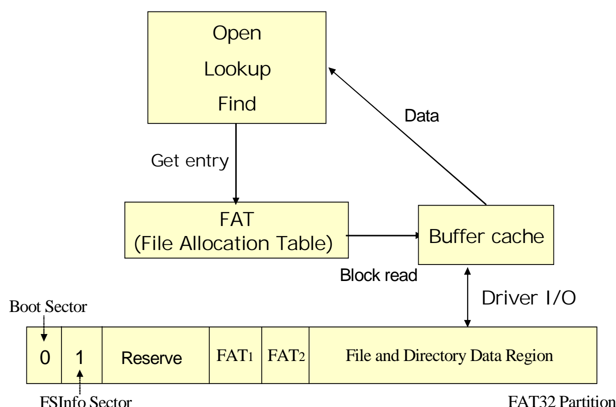


Figure 2: VFAT File System Overview

The bottom block of Figure 2 illustrates the *on-disk format of a FAT32 Partition* [30]. There are Boot sector, Information sector, FAT, File and Directory Data Region in the FAT32 partition. Especially, the root cluster number and cluster size are defined in the boot sector.

2.3 Virtual File System Data Structures

There are multiple data structures and their common operations of VFS, including superblock, file, dentry, inode. File systems should support these data structures and provide their related operations to VFS, such that a modular file system can be maintained under Linux. Figure 3 illustrates the cooperation of vfat FileSystem and VFS. Figure 3(a) shows the data structures of VFS when mounting a vfat FileSystem and Figure 3(b) shows the data structures of VFS when opening a file via Linux vfat FileSystem.

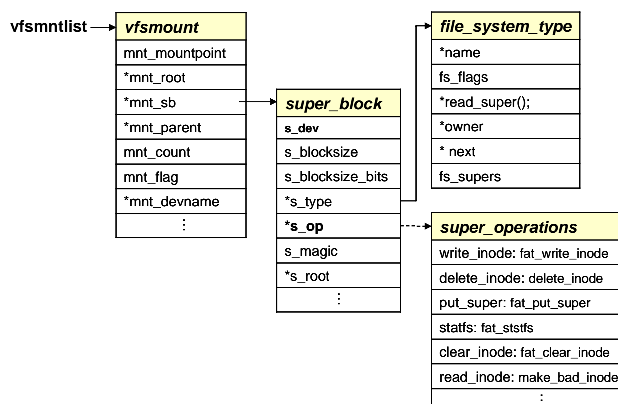
(a) Mounting Linux vfat FileSystem

Linux file systems are implemented as loadable modules. They must be mounted for use and registered at boot time. When mounting, VFS uses a link-list, *vfsmntlist*, to link those mounted file systems, and uses *vfsmount* to describe those mounted file systems. Each mount point has a *super_block* to hold the whole file system's information and status. This *super_block* is filled out by copying the physical partition information, which calls *read_super()* routine defined in *file_system_type*. Finally, the root inode is allocated and subsequent manipulation of the whole file system status should refer to the *super_block* operation methods defined in *super_operations*.

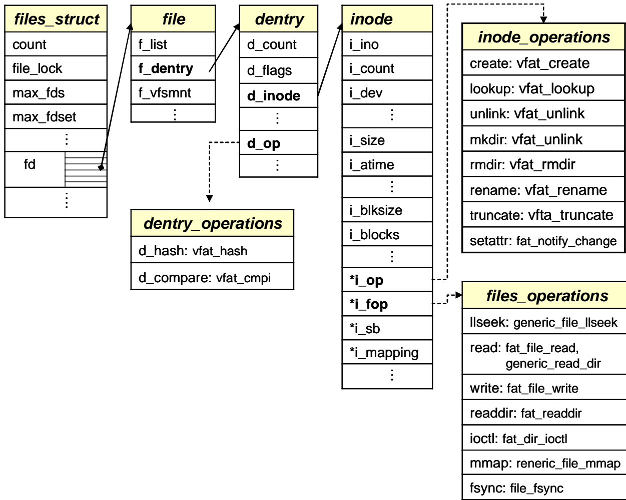
(b) Opening a file via Linux vfat FileSystem

Inode is the abstraction for a *file*. When opening a *file,* file system looks up its inode via *lookup* method defined in *inode_operations*, which should be *vfat_lookup()* in vfat file system. If *lookup* succeeds, the pathname and *inode* structure are then translated to *dentry* structure. Finally, the *file* structure is created and the kernel returns a non-negative integer indexed in the file descriptor table. This non-negative integer can be used for subsequent I/O operations on this file, and its operation methods are defined in *file_operations*. When a file is successfully opend, the non-native integer indexed in the file descriptor table points to the struct *file*. Each *file* structure points to a *dentry* via *file->f_dentry*, in turn, each *dentry* points to an *inode* via *dentry->d_inode*.

To sum up, VFS holds the whole file system's information in *super_block*, and uses *file* for users to maintain file system data. In addition, VFS uses *inode* to store a file's information in file system and uses *dentry* translated from inode to cache frequently used directory entries. Each type of file system has its specific operation methods for *superblock*, *file*, *inode*, and *dentry*, which are defined in *super_operations*, *file_operations*, *inode_operation*, and *dentry_operation* data structures.



(a) Mounting a vfat FileSystem



(b) Opening a file via Linux vfat FileSystem

Figure 3: Virtual File System Data Structures

3. ADAPTATION ISSUES

In this section, we first briefly describe the LyraOS architecture. Then the difficulties in this adaptation work are discussed.

3.1 *LyraOS*

LyraOS [3-7,18,19,25,26] is a component-based operating system which aims at serving as a research vehicle for operating systems and providing a set of well-designed and clear-interface system software components that are ready for Internet PC, hand-held PC, embedded systems, etc.

It was implemented mostly in C++ and some assembly codes. It is designed to abstract the hardware resources of computer systems, such that low-level machine dependent layer is clear cut from higher-level system semantics. Therefore, it can be easily ported to different hardware architectures [4,6]. Each system component is complete separate, self-contained, and highly modular. So the system is also designed to be scalable and reconfigurable.

Besides being light weight system software, it is a time-sharing multi-threading kernel. Threads can be dynamically created and deleted, and thread priorities can be dynamically changed. It provides a preemptive prioritized scheduling and supports various mechanisms for passing signals, semaphores, and messages between threads. On top of the kernel core component, a micro window component with Windows OS look and feel is provided [18]. Figure 4 shows the system architecture.

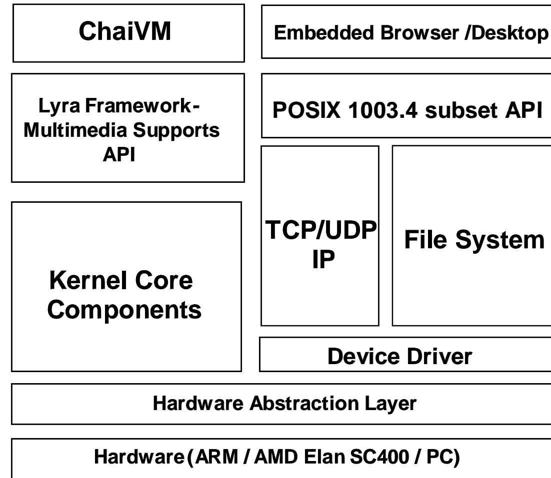


Figure 4: LyraOS System Architecture.

3.2 Adaptation Issues and Difficulties

Since LyraOS and Linux are different in system architecture, Linux vfat FileSystem must be modified for being adapted to LyraOS. LyraOS should also provide some kernel support functions for this adaptation.

Currently, LyraOS supports single address space [4,10] with static binding of applications and kernel. No system call invocation is needed for applications to use the kernel's exported services. To provide the compatible system call interface with other operating systems, LyraOS should provide the same file system interfaces for applications' use as Linux C library functions, as shown in Table 1.

Aside from the different OS architecture, under the component design principle, each LyraOS system component is complete separate, self-contained, and highly modular. Each component has clean exported and imported interfaces for components to communicate with. So, the File System should also be implemented as a separate component such that the advantages of modularity, reconfigurability, component replacement and reuse can be maintained. However, Linux is a monolithic kernel, its File Systems codes have closely relationship with the other kernel function such as device drivers, buffer cache, and kernel core. Further more, although Linux File Systems codes have been modulated under the common interface of VFS, Linux File Systems still closely integrated with VFS. However, LyraOS is a resource limited embedded operating system, integrating Linux VFS into LyraOS File System would involve a lot of data structures and some of them are combined with Linux kernel and can be eliminated at resource limited embedded systems.

So, in order to have a clear cut File System component for LyraOS, we remodel Linux vfat FileSystem and modify some VFS codes as C++ objects, and replace Linux kernel supported functions invocation with equivalent LyraOS kernel components invocation. LyraOS vfat FileSystem's object relation model is shown in Figure 5. The detailed description of these objects and their correlation is presented in Section 4.

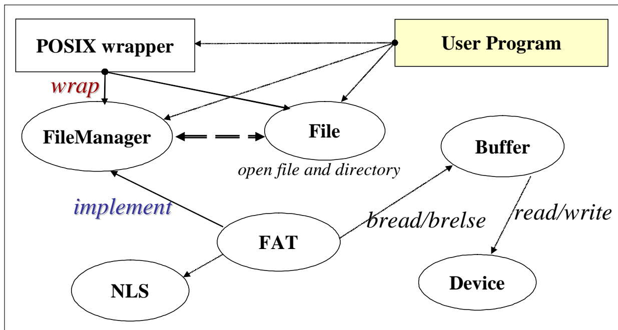


Figure 5: Object Relation

Because Linux vfat FileSystem is a disk-based file system, there is no data path to a RAM-based storage device. To support a RAM-based file system in LyraOS, the driver performs I/O operations between LyraOS File System and RAM should be added. The buffer cache which is used to gain better performance of disk-based file systems should be eliminated at a RAM-based file system to get rid of extra data copy between storage area and buffer cache [8].

To sum up, this adaptation work focused on three parts. First, to implement the vfat FileSystem as a self-contained component, we should clarify its import and export interfaces clearly. Second, we should replace Linux kernel support functions invocation with equivalent LyraOS kernel components invocation. Third, to support RAM-based file systems, data path to RAM should be implemented and their interaction with the buffer cache should be removed from RAM-based file systems.

4 DESIGN AND IMPLEMENTATION

This section presents our adaptation works, including remodeling and modifying Linux vfat FileSystem codes, replacing kernel support functions invocation, and adding data path to RAM-based devices.

4.1 LyraOS vfat FileSystem – Object Relation Model Overview

Figure 5 shows the six sub-components in LyraOS file system. They are *File*, *FileManager*, *FAT*, *NLS*, *Buffer*, and *Device*.

- User programs can request file system services via the POSIX interface or *FileManager* that is the file system interface of LyraOS vfat FileSystem.
- *File* is a friend object of *FileManager*. An user's request of file operations will be past to *FileManager*.
- *FAT* inherits *FileManager* to actually carry out file system services.
- *NLS (National Language Support)* supports *FAT* to store long name in Unicode.
- *Buffer* is an intermediary facility between *FAT* and *Device,* which caches recently accessed device blocks based on LRU policy.
- *Device* provides data paths to disks and RAM, however, devices should be formatted in FAT partition type.

Besides, the *POSIX* wrapper provides the POSIX interface for file system operations.

4.2 Remodeling and modification

Since LyraOS is designed for resource limited embedded systems, to have a modular and self-contained component for LyraOS file system, we remodel and modify Linux vfat Filesystem codes as C++ objects. The remodeling and modification work is presented below:

(a) Eliminating parts of Linux FileSystem

We eliminate some unnecessary mechanisms of Linux file system, such as register/unregister file systems, mount/unmount file systems, lock/unlock objects, permission policy, and others that are combined with Linux kernel, etc. In this way, the total code size of file system can be largely reduced.

(b) FAT

The *inode* data structure is replaced with *fat_node,* as shown in Table 2. The *super_block*, operation methods defined in *super_operations,* and *inode_operations* are integrated into "**class FAT**", as shown in Table 3. The *read_super()* operation is replaced with *FAT::FAT()* constructor. The

fat_node data structure is dedicated to VFAT file system, and *fat_node* cache is implemented as a link-list with no hash table applied. The dentry is not implemented, and directory cache is eliminated. The usage of cache is the tradeoff between memory consumption and performance.

Table 2: fat_node

```
struct fat_node
{
    ino_t ino;          // unique number
    int                 cluster;      // start cluster
    int                 nlink;
    unsigned int        count;        // used count
    unsigned int        attr;         // entry attribute
    unsigned int        state;        // (dirty, lock ... )
    size_tsize;
    time_t              atime;        // access time
    time_t              mtime;      // modified time
    time_t              ctime;        // creation time
    time_t              ctime_ms;
    struct fat_node *next;            // point to next node
};
```

Table 3: class FAT:public FileManager

Operations			**Data**
• *Super operation*	• *File operation*	• *Basic operation*	- cluster_size
- FAT()	- **file_read()**	- vfat_find()	- fats
- ~FAT()	- **file_write()**	- vfat_lookup()	- fat_bits
- fat_dir_size()	- fat_truncate()	- lookup()	- fat_start
- read_node()	- do_truncate()	- vfat_create_entry()	- fat_length
- write_node()	- **truncate()**	- vfat_create()	- dir_start
- get_node()	- **close()**	- vfat_create_a_dotdir()	- dir_entries
- get_node()	• *Dir operation*	- vfat_create_dotdirs()	- data_start
- put_node()	- fat_ino()	- vfat_empty()	- clusters
• *Long name*	- fat_readdirx()	- vfat_free_ino()	- root_cluster
- vfat_valid_shortname()	- dir_read()	- vfat_remove_entry()	- fsinfo_offset
- vfat_valid_longname()	• *File Allocation Table*	- open_namei()	- free_clusters
- vfat_find_form()	- fat_access()	- **open(2)**	- fat_lock
- vfat_format_name()	- fat_free()	- **open(3)**	- prev_free
- vfat_create_shortname()	- fat_add_cluster()	- cp_stat()	- fat_wait
- vfat_find_free_slots()	- fat_get_cluster()	- **stat()**	- node_list
- vfat_fill_long_slots()	- fat_smap()	- do_unlink()	• *Reference interface*
- vfat_build_slots()	- fat_get_entry()	- **unlink()**	
• *misc*	- fat_subdirs()	- do_mkdir()	- *nls
- is_fat32()	• *Date*	- **mkdir()**	
- is_fat16()	- date_dos2unix()		
- is_fat12()	- fat_date_unix2dos()		

(c) File

Some data of *struct File* and methods of *file_operations* are integrated into "**class File**", as shown in Table 4. However, its actual operations should refer to *FileManager* presented at next paragraph.

Table 4: class File

Operations	**Data**
• *Exported interface for user*	- *node
- read()	- mode
- write()	- pos
- truncate()	
- close()	• *Reference Interface*
- readdir()	- FileManager *fm
• *Constructor*	
- File()	
• *Access control function*	
- can_read()	
- can_write()	
- is_dirty()	
- mark_dirty()	

(d) Virtual File System

Linux Virtual File System likely-hood abstraction layer is implemented via virtual functions at "**class FileManager**". Table 5 shows some virtual functions of *FileManager*, and Table 6 shows the "class FileManager". *FAT* is a derived class of *FileManager*, it inherits *FileManager*'s interface and overrides necessary virtual functions to carry out actually file system services.

Table 5: FileManager's Virtual Functions

```
virtual File *creat(const char *pathname, mode_t mode);
virtual File *open(const char *pathname, int flags);
virtual int close(File *file);
virtual int unlink(const char *pathname);
virtual int mkdir(const char *pathname);
virtual int stat(const char *filename, struct stat *buf);
```

Table 6: class FileManager

Operations	**Data**
• Exported interface for user	- *root
- creat()	- *pwd
- open()	
- stat()	- s_blocksize
- unlink()	- s_blocksize_bits
- mkdir()	- s_flags
• Exported interface for File	
- file_read()	- friend class File
- file_write()	
- dir_read()	• Reference interface
- truncate()	- Device * const dev
- close()	

(e) National Language Support

To support storing long name entries in Unicode, the National Language Support (NLS) codes are integrated into “**class NLS**”, as shown in Table 7.

Table 7: class NLS

Operations	Data
- NLS() - ~NLS() - utf8_mbtowc() - utf8_mbstowcs() - utf8_wctomb() - utf8_wcstombs() - to_unicode() - to_charset() **- class CodePage437 : public NLS** - CodePage437() **- class ISO8859_1 : public NLS** - ISO8859_1() - ~ISO8859_1()	- utf8_table utf8_table[] - page_uni2charset - charset2uni

(f) Buffer Cache

Buffer cache’s operations are integrated in “**class Buffer**”, and *struct buffer_head* of Linux kernel is reduced to *struct buffer* for LyraOS, as shown in Table 8. Buffer cache uses device and block number to index the cache entries. However, the hash table used to provide cache entries is removed here. Again, the usage of cache is the tradeoff between memory consumption and performance.

Table 8: class Buffer and struct buffer

Operations	Data	struct buffer

- Buffer()
- ~Buffer()
- getblk()
- fill()
- bread()
- brelse()
- flush()
	- buffers[NR_BUFFERS]
	- *firstbuf
		{
		struct buffer *b_next;
		Device *b_dev;
		char *b_data;
		unsigned int b_flag;
		unsigned int b_count;
		unsigned long b_blkno;
		unsigned long b_size;
		};

(g) POSIX

The POSIX wrapper of LyraOS vfat File System provides the following functions: creat, open, close, unlink, ftruncate, fstat, read, write, opendir, readdir, and closedir. The file manipulation functions, such as open, read/write, and close, need a non-negative integer returned value for file operations. In Linux File System, the file descriptor table, *files_struct*, used to hold the file descriptors is reduced to a File array in LyraOS vfat FileSystem. It is declared as “static File *files[OPEN_MAX];”. File descriptors indexed in the file array point to each opened file.

4.3 Implementing data path to Disk and RAM

(a) Disk

The LyraOS disk-based vfat FileSystem should interact with the disk driver for transferring data. LyraOS provides the Linux device driver emulation environment [26]. Under this environment, Linux device driver codes can be integrated into LyraOS without modification. In LyraOS, a thread is created for running this disk device driver emulation environment. Detailed implementation of this emulation environment is out of the scope of this paper and can be referred to the paper [26].

(b) RAM

Like a FAT32 disk-partition, a single large block is “malloced” in memory as the storage area of the RAM-based vfat FileSystem. Based on this malloced area, the read/write pointer will offset to the requested physical data address according to the file system’s determined block size and block number. The data path to RAM-based storage is implemented at “**class RamDrv**” shown in Table 9.

Table 9: class RamDrv

Operations	Data
Device() ~Device bread()	TOTAL_SECTORS SECTOR_SIZE MemoryBlk size

4.4 LyrsOS Kernel support Modules

To integrate Linux vfat FileSystem into LyraOS, the corresponding Linux kernel support functions invocation should be replaced with the equivalent LyraOS kernel components invocation. Therefore, for this integration, the following functions should be supported in LyraOS kernel:

- Memory Management Component
  - malloc(), free().
- Console I/O Component
  - printf().
- Kernel Core Component: String
  - strcpy(), strlen(), strcmp(), memcpy(), …etc.
- Timer Management Component
  - jiffies.
- Device Driver Component: IDE driver
  - ide_hd_read(), ide_hd_write().

4.5 Interacting with LyraOS vfat FileSystem

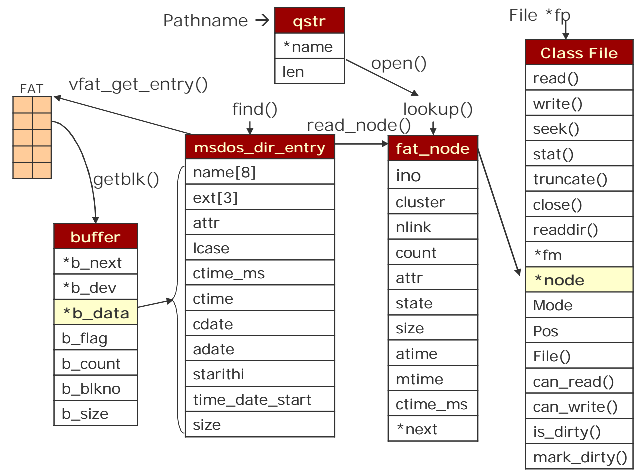
Table 10 illustrates an example of using LyraOS vfat FileSystem, which performs the following operations:

1. Constructing an instance of Device class
2. Constructing an instance of Buffer class
3. Constructing an instance of file system class in application.
4. Using the instance of file system as a FileManager* and requesting system services via its exported interface.
5. Performing File operations or directory operations.

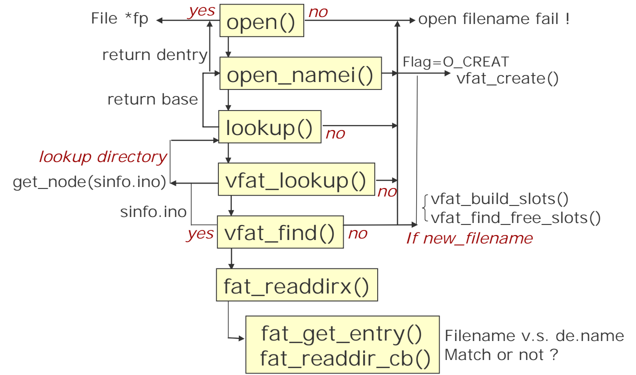
Table 10: Example of Using LyraOS vfat FileSystem

```
Device dev("hda1");
BufferManager *bufmgr = new BufferManager();
FileManager *fs = new FAT(&dev);
File *fp = fs->open(src, O_RDONLY);
if (fp) {
        while ((len = fp->read(buf, sizeof(buf)))) {
                // output content to screen
        }
        fp->close();
}
delete fs;
delete bufmgr;
```

When a file is opened via LyraOS vfat FileSystem, the involved data structure and data flow are illustrated in Figure 6(a), and its operation flow chart is illustrated in Figure 6(b). As shown in the illustration, the data structures and operation functions of Linux vfat FileSystem are largely simplified after being adapted to LyraOS.



(a) Data Structures and Data Flow



(b) Operation Flow Chart

Figure 6: Opening a File in LyraOS vfat FileSystem

5 PERFORMANCE EVALUATIONS

This section presents the LyraOS RAM-based vfat FileSystem performance evaluation. Under Andrew File System BenchMark files [32], the performance of pre-creation files, mkdir, read, copy, and unlink operations are measured.

Andrew BenchMark provides various size data, ranging from 23 bytes to 37 KB. We use these data to perform this measurement, including: pre-creating files (BenchMark data will be precreated in memory), making directories (mkdir), reading files (read), copying file (copy), and deleting files (unlink). However, in order to apply this measurement to LyraOS vfat FileSystem, which is a static binding embedded system without system call invocation, the Andrew BenchMark is modified to adapt to our measurement by running LyraOS vfat FileSystem to perform the above four phases as separate programs. The data files should be pre-created in memory for experiments.

The Experimental platform is a PC with Pentium-III 1GHz processor and 128MB RAM. This evaluation is performed at LyaOS RAM-based vfat FileSystem under two circumstances. One is the file system with buffer cache, and the other without buffer cache.

Figure 7 shows that LyraOS vfat FileSystem on RAM-based storage without buffer cache would perform 25% better than the same file system with buffer cache applying. This result also shows that our modified vfat FileSystem occupies only 75% of the total cost of file system operations. This performance gain results from eliminating extra data copy overhead between storage area and buffer cache. This means that buffer cache is unnecessary for RAM-based file systems and should be removed in RAM-based file systems.

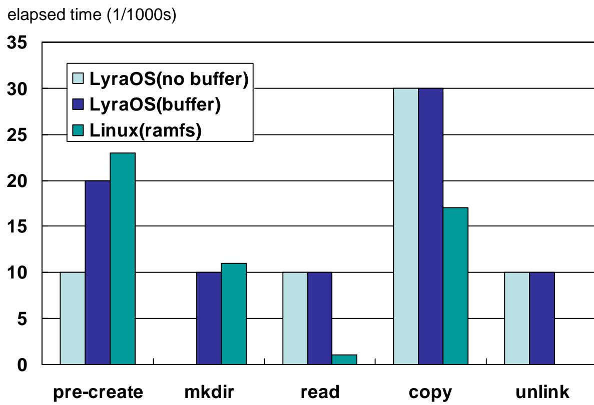


Figure 7: Performance Comparison of Various RAM-based File Systems

For the performance comparison of LyraOS RAM-based vfat FileSystem without buffer cache with Linux ramfs [29], the evaluation program ran in LyraOS is unmodified to Linux ramfs's evaluation except that mkdir is replaced with a script program to do the mkdir system calls. It can be seen from Figure 7 that Linux ramfs incurs more elapsed time than LyraOS RAM-based vfat without buffer cache at the pre-creation and mkdir phases. Although under Linux VFS's Cache mechanisms, ramfs performs better than LyraOS at read and copy phases. However, if referring to pre-creation phase, the create operation is almost the write operation, and the Cache mechanisms are not involved in ramfs at this phase. This implicates that when Linux VFS's Cache mechanisms are not applied, our write performance is better than ramfs. However, this is a tradeoff between memory consumption and system performance.

Aside from the usage of various caches such as directory cache, inode cache for performance improvement, Linux ramfs uses mmap mechanism and page cache to optimize its operation [29]. As a result, ramfs is tightly correlated with Linux kernel's memory management mechanism, which is against the component design principle and ramfs is not able to be a separate component for adapting into a target system.

6 CONCLUDSIONS

To add data storage capability to LyraOS, we have successfully adapted Linux vfat FileSystem to LyraOS. Aside from disk-based vfat FileSystem, the RAM-based vfat FileSystem is also implemented. In this paper, we have described how to solve the adaptation difficulties from different system design principles and different kernel architectures. The adaptation work focuses on three parts as follows. To implement the vfat filesystem as a self-contained component, we clarify its import and export interfaces and modify Linux vfat file system codes to separate them from the other Linux kernel functions. Some kernel support modules invocations are replaced with specific LyraOS kernel components invocation. Data path to RAM-based device is added, and compatible file system interfaces (POSIX) of LyraOS vfat filesystem is implemented.

In addition to implement file system as a self-contained component, this adaptation work eliminates extra data copy of buffer cache, and simplifies or removes lots of Linux file system data structures.

Performance evaluation under modified Andrew BenchMark shows that LyraOS RAM-based vfat FileSystem without buffer cache occupies only 75% of the total file system operations, and the remainder is the cost for data copy to and from buffer cache. While comparing our modified vfat FileSystem based on RAM without buffer cache with Linux ramfs, it shows that our modified vfat FileSystem is efficient and can adapt to embedded systems.

For disk-based LyraOS vfat FlieSystem performance evaluation, Linux driver emulation environment is well implemented in LyraOS for reusing Linux device drivers [26]. We are currently under the work to replace the IDE (Integrated Device Electronics) driver with the current version of Linux drivers and conduct the performance comparison of LyraOS vfat FlieSystem with Linux

vfat FlieSystem.

To sum up, the success of this porting and the experience of this integration study can be of practical value to serve as the reference for embedding Linux file system into target systems that need storage capability.

7. REFERENCES

[1] Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner, *Linux Kernel Internals,* Addison-Wesley Publishing Company Inc., September 1996.

[2] J. Bruno, J. Brustoloni, E. Grabber, A. Silberschatz, and C. Small, "Pebble: A Component Based Operating System for Embedded Applications," In *Proceedings of 3rd Symposium on Operating Systems Design and Implementation*, USENIX, February 1999.

[3] Zan-Yu Chen, "Draft-LyraOS Component Interface Design Spec. – Machine Dependant Layer, Ver. 1.2," *Technical Report*, Department of Information and Computer Science, National Chiao-Tung University, 2000.

[4] Zan-Yu Chen, "A Component Based Embedded Operating System," *Master Thesis*, Department of Information and Computer Science, National Chiao-Tung University, June 2000.

[5] Zan-Yu Chen, Mei-Ling Chiang, and Ruei-Chuan Chang, "The LyraOS APIs," *Technical Report,* Department of Information and Computer Science, National Chiao-Tung University, 2000.

[6] Zan-Yu Chen, Mei-Ling Chiang, and Ruei-Chuan Chang, "Putting LyraOS onto ÉlanTM µforCE," *Technical Report*, Department of Information and Computer Science, National Chiao-Tung University, 2000.

[7] Mei-Ling Chiang, "Draft-LyraOS Component Interface Design Spec. - Kernel Core, Ver. 1.7," *Technical Report*, Department of Information Management, National Chi-Nan University, 2000.

[8] McKusick MK, Joy WN, Leffler SJ, Fabry RS, "A Pageable Memory Based FileSystem," *Proceedings of USENIX Conference*, June 1990.

[9] An-I Andy Wang, Geoffrey H. Kuenning, Peter Reiher, Gerald J. Popek, "Conquest: Better Performance Through A Disk/Persistent-RAM Hybrid File System," *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, Monterey, January 2002.

[10] Luke Deller and Gernot Heiser, "Linking Programs in a Single Address Space," In *Proceedings of 3rd Symposium on Operating Systems Design and Implementation*, USENIX, February 1999.

[11] Embedded Configurable Operating System (eCos), http://www.redhat.com/products/ecos/, 2000.

[12] EPOC, Psion, Nokia, Ericsson, and Motorola, http://www.tcm.hut.fi/Opinnot/Tik-111.550/1999/Esitelmat/Symbian/report.html, 2000.

[14] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. "The Flux OSKit: A Substrate for OS and Language Research," In *Proc. Of the 16th ACM Symp. On Operating System Principles*, Oct. 1997.

[15] GNU General Public License (GPL), http://www.linux.org/info/gnu.html.

[16] GEOS, Geoworks Corporation, http://www.geoworks.co.uk/os/wireless_big.html.

[17] E. Grabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz, "The Pebble Component-Based Operating System," In *1999 USENIX Annual Technical Conference*, June 1999.

[18] Wen-Shu Huang and R. C. Chang, "An Implementation of a Configurable Window Systemon LyraOS," *Master Thesis*, Department of Computer and Information Science, National Chiao Tung University, 2000.

[19] Chi-Wei Yang, C. H. Lee, and R. C. Chang, "*Lyra:* A System Framework in Supporting Multimedia Applications," *IEEE International Conference on Multimedia Computing and Systems'99 (ICMCS'99)* Florence, Italy, June 1999.

[20] X. Liu, C. kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable, "Building reliable, high-performance communication systems from components," In 17th *ACM Symposium on Operating Systems Principles*(SOSP' 99), Dec. 1999.

[21] MicroC/OS II homepage at http://www.ucos-ii.com/, 2000.

[22] PalmOS homepage at http://www.palmos.com/, 2000.

[23] QNX homepage at http://www.qnx.com/, 2000.

[24] Windows CE homepage at http://www.microsoft.com/embedded/, 2000.

[25] Jer-Wei Chuang, Kim-Seng Sew, Mei-Ling Chiang, and Ruei-Chuan Chang, "Integration of Linux Communication Stacks into Embedded Operating

Systems," *Proceeding of the 2000 International Computer Symposium*, Taiwan, 2000.

[26] Chi-Wei Yang, Paul C. H. Lee, and R. C. Chang, "Reuse Linux Device Drivers in Embedded Systems," *Proceeding of the 1998 International Computer Symposium (ICS'98)*, Taiwan, 1998.

[27] Ricardo Galli, "Journal File Systems in Linux," *UPGRADE*, Vol. II, No. 6, December 2001, pp. 50-56.

[28] The Linux Kernel at http://www.tldp.org/LDP/tlk/tlk.html, 2002.

[29] Linux Kernel 2.4 Internals at http://www.tldp.org/LDP/lki/index.html, 2002.

[30] Microsoft Corporation, "FAT: General Overview of On-Disk Format," Version 1.03, December 6, 2000.

[31] Z. Y. Cheng, M. L. Chiang, and R. C. Chang (2000), "A component based operating system for Resource limited embedded devices," *IEEE International Symposium on Consumer Electronics (ISCE'2000)*, HongKong, Dec. 5-7, 2000.

[32] Howard J, Kazar M, Menees S, Nichols D, Satyanarayanan M, Sidebotham R, West M. Scale and Performance in a Distributed File System, *ACM Transactions on Computer Systems*, 6(1), pp. 51-81, February 1988.